Dodekalogue

wiki.tcl-lang.org/page/Dodekalogue

Dodekalogue is the common name for the twelve rules that define the syntax and semantics of Tcl.

See also

official documentation

Every word is a constructor
An argument for typed values.

{*}
Argument expansion.

Tcl syntax

Unix Shells
Comparison and contrast between Unix shell and Tcl syntax.

BNF for Tcl
or *Why There's no BNF for Tcl*.

Tcl Rules Redux
An alternate description of the rules of Tcl.

Description

Prior to the introduction of the twelfth rule, Argument expansion, the rules were known as the Endekalogue.

KBK, in a silly mood, points out that the *real* Dodekalogue is over at http://www.faqs.org/rfcs/rfc1925.html - and observes that Tcl/Tk already has one of the best implementations of RFC 1925 available.

Script Substitution is a more apt name for **command substitution** since brackets can contain entire scripts, not just an individual command. As with any script, semicolons and newlines can be used to separate multiple commands. The result of the script is the result of the final command. A new stack frame is not created, so using return or break or the like will cause the caller to return, etc.

AMG: Comments may be used within the bracketed script, but take care that the closing bracket isn't swallowed up by the comment! In other words, the comment must end with a newline (itself not having an odd number of backslashes preceding) before a close bracket is recognized as such.

Octalogue

The official rules of Tcl are described in the **Rules** section further down this page. This section presents a more concise, but slightly more complete description of the rules. In this description, *brackets* means **[** and **]** pairs, and *braces* means **{** and **}** pairs. quotes means **"**.

Script
A script is composed of commands delimited by newlines or semicolons, and a command is composed of words delimited by whitespace.

Evaluation
Substitutions in each word are processed, each word that begins with **{*}** is expanded into multiple words in the command, and the first word of each command is used to locate a routine, which is then called with the remaining words as its arguments.

Comment
If **#** is encountered when the name of a command is expected, it and the subsequent characters up to the next newline are ignored, except that a newline preceded by an odd number of **** characters does not terminate the comment.

$*varname*
Replaced by the value of the variable named *varname*, which is a sequence of one or more letters, digits, or underscore characters, or namespace separators, optionally followed by any sequence of characters enclosed in braces, upon which substitutions are performed to obtain a second name, which is name of a variable in an array. If *varname* is enclosed in braces, the variable name, including optional second name in parentheses, is composed of all the characters between the braces, and no substitution is performed on the name or second name.

\*char*
Replaced with the character following *char*, except when char is **a**, **b**, **f**, **n**, **t**, **r**, **v**, which respectively signify the unicode characters audible alert (7), backspace (8), form feed (c), newline (a), carriage return (d), tab (9), and vertical tab (b), respectively. Additionally, when **char** is **o**, **x**, **u**, or **U**, it represents a unicode character by 1 to 3 octal digits, 2 hexadecimal digits, 1 to 4 hexadecimal digits, or 1 to 8 hexadecimal digits, respectively.

Brackets
A script may be embedded at any position of a word by enclosing it in brackets. The embedded script is passed verbatim to the interpreter for execution and the result is inserted in place of the script and its enclosing brackets. The result of a script is the result of the last routine in the script.

Quotes
In a word enclosed in quotes, whitespace and semicolons have no special meaning.

Braces

In a word enclosed in braces, whitespace, semicolons, **$**, brackets, and **** have no special meaning, except that *****newline* substitution is performed when the newline character is preceded by an odd number of **** characters. Any nested opening brace character must be paired with a subsequent matching closing brace character. An opening or closing brace character preceded by an odd number of **** characters is not used to pair braces.

Rules

The following rules define the syntax and semantics of the Tcl language:

[1] Commands

A Tcl script is a string containing one or more commands. Semicolons and newlines are command separators unless quoted as described below. Close brackets are command terminators during command substitution (see below) unless quoted.

[2] Evaluation

A command is evaluated in two steps. First, the Tcl interpreter breaks the command into words and performs substitutions as described below. These substitutions are performed in the same way for all commands. The first word is used to locate a routine to carry out the command, then all of the words of the command are passed to the routine. The routine is free to interpret each of its words in any way it likes, such as an integer, variable name, list, or Tcl script. Different commands interpret their words differently.

[3] Words

Words of a command are separated by white space (except for newlines, which are command separators).

[4] Double quotes

If the first character of a word is double quote (“**"**”) then the word is terminated by the next double quote character. If semicolons, close brackets, or white space characters (including newlines) appear between the quotes then they are treated as ordinary characters and included in the word. Command substitution, variable substitution, and backslash substitution are performed on the characters between the quotes as described below. The double quotes are not retained as part of the word.

[5] Argument expansion

If a word starts with the string “**{*}**” followed by a non-whitespace character, then the leading “**{*}**” is removed and the rest of the word is parsed and substituted as any other word. After substitution, the word is parsed as a list (without command or variable substitutions; backslash substitutions are performed as is normal for a list and individual internal words may be surrounded by either braces or double-quote characters), and its words are added to the command being substituted. For instance,

```
cmd a {*}{b [c]} d {*}{$e f {g h}}
```

is equivalent to

```
cmd a b {[c]} d {$e} f {g h}
```

[6] Braces

If the first character of a word is an open brace (“**{**”) and rule [5] does not apply, then the word is terminated by the matching close brace (“**}**”). Braces nest within the word: for each additional open brace there must be an additional close brace (however, if an open brace or close brace within the word is quoted with a backslash then it is not counted in locating the matching close brace). No substitutions are performed on the characters between the braces except for backslash-newline substitutions described below, nor do semi-colons, newlines, close brackets, or white space receive any special interpretation. The word will consist of exactly the characters between the outer braces, not including the braces themselves.

[7] Command substitution (script substitution)

If a word contains an open bracket (“**[**”) then Tcl performs command substitution. To do this it invokes the Tcl interpreter recursively to process the characters following the open bracket as a Tcl script. The script may contain any number of commands and must be terminated by a close bracket (“**]**”). The result of the script (i.e. the result of its last command) is substituted into the word in place of the brackets and all of the characters between them. There may be any number of command substitutions in a single word. Command substitution is not performed on words enclosed in braces.

[8] Variable substitution

If a word contains a dollar-sign (“**$**”) followed by one of the forms described below, then Tcl performs variable substitution: the dollar-sign and the following characters are replaced in the word by the value of a variable. Variable substitution may take any of the following forms:

$*name*
Name is the name of a scalar variable; the name is a sequence of one or more characters that are a letter, digit, underscore, or namespace separators (two or more colons). Letters and digits are only the standard ASCII ones (**0-9**, **A-Z** and **a-z**)

$*name*(*index*)
Name gives the name of an array variable and *index* gives the name of an element within that array. *Name* must contain only letters, digits, underscores, and namespace separators, and may be an empty string. Letters and digits are only the standard ASCII ones (**0-9**, **A-Z** and **a-z**). Command substitutions, variable substitutions, and backslash substitutions are performed on the characters of *index*.

${*name*}

Name is the name of a scalar variable or array element. It may contain any characters whatsoever except for close braces. It indicates an array element if name is in the form "*arrayName*(*index*)" where *arrayName* does not contain any open parenthesis characters, "**(**", or close brace characters, "**}**", and index can be any sequence of characters except for close brace characters. No further substitutions are performed during the parsing of name.

There may be any number of variable substitutions in a single word. Variable substitution is not performed on words enclosed in braces.

Note that variables may contain character sequences other than those listed above, but in that case other mechanisms must be used to access them (e.g., via the single-argument form of set).

[9] Backslash substitution

If a backslash ("****") appears within a word then backslash substitution occurs. In all cases but those described below the backslash is dropped and the following character is treated as an ordinary character and included in the word. This allows characters such as double quotes, close brackets, and dollar signs to be included in words without triggering special processing. The following table lists the backslash sequences that are handled specially, along with the value that replaces each sequence.

\a
Audible alert (bell) (0x7).

\b
Backspace (0x8).

\f
Form feed (0xc).

\n
Newline (0xa).

\r
Carriage-return (0xd).

\t
Tab (0x9).

\v
Vertical tab (0xb).

\<newline>*whiteSpace*
A single space character replaces the backslash, newline, and all spaces and tabs after the newline. This backslash sequence is unique in that it is replaced in a separate pre-pass before the command is actually parsed. This means that it will be replaced even

when it occurs between braces, and the resulting space will be treated as a word separator if it isn't in braces or quotes.

\
Backslash (“\”).

\ooo
The digits *ooo* (one, two, or three of them) give an eight-bit octal value for the Unicode character that will be inserted. The upper bits of the Unicode character will be 0.

\x*hh*
The hexadecimal digits *hh* give an eight-bit hexadecimal value for the Unicode character that will be inserted. Any number of hexadecimal digits may be present; however, all but the last two are ignored (the result is always a one-byte quantity). The upper bits of the Unicode character will be 0.

\u*hhhh*
The hexadecimal digits *hhhh* (one, two, three, or four of them) give a sixteen-bit hexadecimal value for the Unicode character that will be inserted. The upper bits of the Unicode character will be 0.

\U*hhhhhhhh*
The hexadecimal digits *hhhhhhhh* (one up to eight of them) give a twenty-one-bit hexadecimal value for the Unicode character that will be inserted, in the range U+0000..U+10FFFF. The parser will stop just before this range overflows, or when the maximum of eight digits is reached. The upper bits of the Unicode character will be 0.

The range U+010000..U+10FFFD is reserved for the future.

Backslash substitution is not performed on words enclosed in braces, except for backslash-newline as described above.

[10] Comments.

If a hash character (“**#**”) appears at a point where Tcl is expecting the first character of the first word of a command, then the hash character and the characters that follow it, up through the next newline, are treated as a comment and ignored. The comment character only has significance when it appears at the beginning of a command.

[11] Order of substitution

Each character is processed exactly once by the Tcl interpreter as part of creating the words of a command. For example, if variable substitution occurs then no further substitutions are performed on the value of the variable; the value is inserted into the word verbatim. If command substitution occurs then the nested command is processed entirely by the recursive call to the Tcl interpreter; no substitutions are performed before making the recursive call and no additional substitutions are performed on the result of the nested script.

Substitutions take place from left to right, and each substitution is evaluated completely before attempting to evaluate the next. Thus, a sequence like

```
set y [set x 0][incr x][incr x]
```

will always set the variable y to the value, *012*.

[12] Substitution and word boundaries

Substitutions do not affect the word boundaries of a command, except for argument expansion as specified in rule [5]. For example, during variable substitution the entire value of the variable becomes part of a single word, even if the variable's value contains spaces.

Details

[1] Commands

Technically, even a script that is the empty string contains one command: The empty command.

Double Expansion

Given

```
set q [list [list a b c] x y z]
```

Should {*}{*}$q produce a b c x y z?

(anonymous): Good question; the wording here allows {*}{*}$q, but I am not sure if this is supposed to be allowed.

DGP: No, that should continue to be a syntax error. TIP 157 accepted argument expansion (post-substitution determination of word boundaries upon explicit request), but never proposed anything about double [expansion].

What is Whitespace?

TIP#407 is explicit about which characters are treated as whitespace by the list parser (and consequently, the command parser). This happens to be the intersection of the POSIX Portable Character Set (ASCII) and characters with the UNICODE White_Space property . In Tcl, the intersection of {[string is space] && [string is ascii]} (see [string is]).

\u00A0 is a Unicode whitespace character but outside the POSIX range, so it is not a space for Tcl word-split purposes:

```
% set a a\u00a0b
a b
% llength [ split $a ]
1
% eval "set a a\u00a0b"
a b
```

Inline unicode escapes do not count as word separators (rule 9 is applied after rule 3):

```
% set a a\u0020b
a b
% llength [ split $a ]
2
% eval "set a a\u0020b"
wrong # args: should be "set varName ?newValue?"
```

Backslash-newline inside brace-quoted words

AMG: Why is backslash-newline substitution performed inside brace-quoted words? This has caused trouble for me before. The only justification I could think of was to establish an equivalence between "foo" and subst {foo}, for all possible foo. However, this justification doesn't hold, since subst internally performs backslash-newline substitution.

Let me give an example of where this hurts. In my Wibble web server, the template command performs template substitution like in Templates and subst. One thing it's capable of doing is joining lines separated by backslash-newline, which is certainly a useful feature. However, in the case of joining long lines which are Tcl script (i.e. have a leading %), this feature works differently depending on where the input comes from. If the input is read from a file, there must be a leading % on all lines to be joined, which is consistent with the notion that the lines are all Tcl script. If the input is supplied inside a Tcl script, there must *not* be a leading % on any line but the first, otherwise the %'s get merged in which the script.

This happens because when the input is inside a Tcl script, Tcl joins the lines (% and all) before ever passing the argument to template. When the input comes from a file, the Tcl parser isn't executed until *after* template picks out all the lines that start with % and turns them into a script.

steveb: FWIW, Jim doesn't do this backslash-newline substitution in braces, and it has never caused me a problem.

AMG: So, what is it good for? I found another problem it causes: errorInfo line numbers. Since lines separated by backslash-newline are joined by the parser before they're ever passed to proc, they collectively count as a single line for the purpose of counting line numbers, which obviously differs from how line numbers are counted in text editors.

Surely the execution engine can be made to internally treat backslash-newline as a word separator rather than a command separator. This would remove any need for the parser to ever muck with the contents of braced words. Actually, I'm pretty sure it already has this feature, since backslash-newline works as expected (as a word separator or comment continuation) when the script is at the toplevel, no braces in sight.

AMG: Clarification: backslash-newline-whitespace substitution is only done if there are an odd number of backslashes before the newline.

```
puts {a
b
}
puts {a\
b
}
puts {a\\
b
}
puts {a\\\
b
}
```

The above prints:

```
a
b

a b

a\\
b

a\\ b
```

So yes, it is indeed impossible for a brace-quoted word to have an odd number of backslashes immediately before a newline. Even numbers of backslashes (for example zero!) are fine.

PYK 2016-06-03: Double quotes work the same way, and if you look at the contents as a message to be displayed, the backslash-newline treatment does the unindenting that's probably desired. Without the backslash-newline parsing behaviour of Tcl, there would have to be a lot more [dedent {this message ...}] in scripts. Wibble attempts a clever hack of using the Tcl parser to do double-duty for parsing templates, and it "would have gotten away with it too, if it wasn't for you meddling backslash-newlines!", but that's because Wibble templates aren't really quite Tcl scripts. scriptsplit is likely one workaround.

AMG: Indeed, double quotes do work the same way, but they also process every other kind of substitution so this comes as no surprise. One of the two fundamental differences between double quotes and braces is the presence or absence of substitution. (The other is nesting.)

Anyway, I just came here to point out that I did find a use for backslash-newline inside braces. I'm disinclined to use it since I don't like the feature and want to be prepared should we have the guts to remove it in novem/Tcl 9. In tcltest, it's common to test error messages with something like the following:

```
test demo-1.1 {demonstration of error} -body {
    list [catch {lsearch -x a b} msg] $msg
} -result {1 {bad option "-x": must be -all, -ascii,\
-bisect, -decreasing, -dictionary, -exact, -glob,\
-increasing, -index, -inline, -integer, -nocase, -not,\
-real, -regexp, -sorted, -start, or -subindices}}
```

The backslashes make this work. If they were not substituted, the exact comparison would fail. Using double quotes would be a workaround, but it's a little slice of quoting hell.

```
test demo-1.1 {demonstration of error} -body {
    list [catch {lsearch -x a b} msg] $msg
} -result "1 {bad option \"-x\": must be -all, -ascii,\
-bisect, -decreasing, -dictionary, -exact, -glob,\
-increasing, -index, -inline, -integer, -nocase, -not,\
-real, -regexp, -sorted, -start, or -subindices}"
```

[list] is a safer way to construct the expected result.

```
test demo-1.1 {demonstration of error} -body {
    list [catch {lsearch -x a b} msg] $msg
} -result [list 1 "bad option \"-x\": must be -all,\
-ascii, -bisect, -decreasing, -dictionary, -exact,\
-glob, -increasing, -index, -inline, -integer, -nocase,\
-not, -real, -regexp, -sorted, -start, or -subindices"]
```

Though this still necessitates backslash-quoting the embedded double quotes.

PYK 2017-08-16: scripted list is useful for crafting the -result value for a test. I also hope that the backslash-newline substitution in braces is removed.

AMG: A note about braced backslash-newline replacement. It only happens when parsing commands, not when parsing a list.

```
% eval puts\ \{a\\\nb\}
a b
% lindex puts\ \{a\\\nb\} 1
a\
b
```

PYK 2017-09-30: Another point in favor of abandoning backslash-newline replacement in braces: It's an obstacle to writing multi-line macros in C when using Critcl.

AMG, 2020-07-29: Braced backslash replacement makes line continuation for "Templates and subst" templates inconsistent.

First, consider the following template, just so you're familiar with the syntax:

```
this text is emitted once
% for {set i 1} {$i <= 10} {incr i} {
and this text is emitted ten times (#$i)
% }
```

How does one go about splitting that "for" line? If the template is read in from a file, here's how:

```
this text is emitted once
% for {set i 1} {$i <= 10}\
%     {incr i} {
and this text is emitted ten times (#$i)
% }
```

But if the template is brace-quoted and comes from a script, this must be done instead:

```
this text is emitted once
% for {set i 1} {$i <= 10}\
      {incr i} {
and this text is emitted ten times (#$i)
% }
```

Think about what you get when you use the wrong style for the wrong situation. Using the first (file) style of continuation within braces would result in the continued line being joined before ever getting passed to the template parser, meaning that it ends up with a stray % word in the middle of the for line. On the contrary, using the second (brace) style of continuation with a file would result in the first script ending with a stray backslash, as well as being followed by direct emission of "{incr i} {".

As steveb said above, Jim doesn't replace backslash-newline-whitespace inside braces, and he never missed this feature. The script parsing code already knows how to handle backslash continuation, so there's no need to double up and have the brace quoting code replace backslashes too.

That's not to say braced backslash replacement is never useful. As I said above (years ago), I've needed it for various tcltest cases where I'm trying to match against a long list. I can't use newlines as a delimiter because a literal string comparison is used by default, so instead I can use backslash-newline-whitespace as a surrogate for a single space. However, I could instead use double quotes, a context in which backslash replacement does make sense, but then I'd have to also use backslashes to protect every metacharacter, including backslash itself. My point is that removing this feature would be an incompatible change. Even though the script parsing code knows about backslash replacement, the list parsing code does not.

In my opinion, a better solution would be to construct a list rather than give literal text, but the list command is awkward to use (also needs backslash line continuations as well as backslashing/quoting of metacharacters), so this makes me wish for dedicated list construction syntax, a feature Tcl lacks. But I'm not going to derail this page with detailed, backward-incompatible new syntax proposals.

(Pardon me for repeating several things that were said years ago, didn't go back and review everything before writing my new comments.)

AMG: Normally, according to rule [9], a backslash-newline sequence is replaced by a single space. Here's a way to instead remove it entirely. This is useful for breaking a long word across lines without chopping it into multiple words.

Instead of:

```
puts this_is_a_really_long_word\
    _which_can't_have_spaces
```

Do:

```
puts this_is_a_really_long_word[
    ]_which_can't_have_spaces
```

The trick is to hide the newline inside a null script substitution.

Bonus! You can also embed comments in the null script substitution. However, you can't put the close bracket on the same line as a comment.

At the moment, this approach is not super-efficient, and it can stand to be improved. [tcl::unsupported::disassemble] says:

```
ByteCode 0x000000000261A5E0, refCt 1, epoch 16, interp 0x00000000026562E0 (epoch
16)
  Source "puts this_is_a_really_[\n    ]long_word"
  Cmds 1, src 38, inst 13, litObjs 4, aux 0, stkDepth 4, code/src 0.00
  Proc 0x000000000261D870, refCt 1, args 0, compiled locals 0
  Commands 1:
      1: pc 0-11, src 0-37
  Command 1: "puts this_is_a_really_[\n    ]long_word"
    (0) push1 0         # "puts"
    (2) push1 1         # "this_is_a_really_"
    (4) push1 2         # ""
    (6) push1 3         # "long_word"
    (8) concat1 3
    (10) invokeStk1 2
    (12) done
```

This technique is discussed further on the "[_" page (which is surprisingly difficult to link to).

What is a "word"?

CMcC 2005-02-03:

It seems inconsistent (although not necessarily inconsistent with the above) that while {} [] $ and "" seem to be treated similarly in the above, ${x}y $x(z)y and [x]y are accepted by the tcl parser (and are presumably *words*), but "x"y and {x}y are not.

It would also seem that x"y and x{y are words, and this is explicitly allowed by the requirement that " and { be the first characters of a word to have special interpretation. Nothing in the above seems to explicitly require that " and } be the last characters of a word to have special interpretation, yet this seems to be the case.

I'm interested in why this is, historically, pragmatically, operationally.

I suppose, pragmatically, {x}y is most likely to be a typo for {x} y, and similarly for "x"y. Shouldn't this be made explicit (or implicit) in the dodecalogue? Perhaps it is already, and I've just missed it.

Lars H: It is explicit. The rules for quote- and brace-delimited words explicitly say that the *matching* **"** or **}** *terminate* the word. Since words have to be separated by whitespace, it is then an error if there (in the same command) is some non-whitespace following the terminating quote or brace.

NJG: Then what about this:

```
% set a b
b
(bin) 2 % set c ${a}x
bx
```

(Scroll down for more on the subject from me. If you are interested, that is)

NJG 2005-02-03:

Or look at this from my console

```
% set var#3 9
9
(bin) 2 % set var#3
9
(bin) 3 % set a $var#3
can't read "var": no such variable
(bin) 4 % set a ${var#3}
9
(bin) 5 % set "var#3"
9
(bin) 6 % set a $"var#3"
$"var#3"
(bin) 7 %
```

Tcl accepts any garbage as a variable name in a set statement a lot of which it cannot handle in $ dereferencing!

Peter da Silva: I don't see the problem here. $ dereferencing accepts any garbage set does, by using the ${...} form. (*FB: not* any garbage*, see below*). The ${...} form is not distinct from the $alphanumerics form. THIS WAS DELIBERATE.

DKF: Yes. $foo is really just shorthand for set foo (except it doesn't go through set itself) and the syntactic constraints on $ substitution are useful because they do what you want most of the time. (e.g. $frame.component is very common in Tk code) - RS ... and with bracing you can dollar-evaluate any weird name, too.

NJG 2005-02-04:

Gentlemen, you are missing the point I am making here. It is **not** about **how** and **why**. It is in the **affirmative**: the morphology of Tcl and the behaviour (as of now there are no such thing as rules) of the $ *substitution* must be revised. As they are now implemented, variable and command names you never in your right mind would think of using are accepted in definitive places just to prevent (rather make very awkward) the use of constructed names. When I need indexed variables, sometimes I would rather write just ***$A$i$j*** than **$A($i,$j)** or use just ***$$N*** instead of **set $N**. Moreover, the »forbidden« forms can be implemented more efficiently!

For rule number one we should assert that in **$**<string> the same quoting/substitution rules apply for ***<string>*** as in any other places. To avoid unmanageable situations and not to go head on against the present conventions the character set for a variable or command name should then be reasonably restricted.

The present practice IM**N**HO is just a sloppyness in the interpreter realization that may go back as far as the first implementation but that is no reason to remain unchanged forever.

Lars H: I **strongly** disagree. The ability to use weird names is sometimes tremendously useful, and it's not like it is hard to avoid generating weird names if one doesn't like them. Changing the interpretation of $A$i$j would definitely wreck tons of Tcl code (and your suggested interpretation is inconsistent -- it couldn't be like $A($i,$j), but would rather be like $A($i($j))).

NJG 2005-02-06: OK, I give up. Not that I agree, I just have not enough time right now to engage in a discussion of merit. However, I would really appreciate if somebody gave me a digestible reason why on earth in *$ substitution* **#** should act as a delimiter,

Lars H: Because # is not "a letter, digit, underscore, or namespace separator" (rule 8 above). Variable substitution is designed to be abstemious with respect to what it will grab.

(presumably) NJG: a pair of **"** characters should not keep their role (that is forcing any substitutions in the enclosed text before $ is applied)

Lars H: Because that is not their role! Read rule 4. The special role of the quotes, which they only take on at the **beginning** of a word, is to make whitespace, semicolons, and close brackets count as characters in the word rather than as word separators. Period. Quoted and unquoted words behave essentially the same. It is always possible to get to the exact same end with a word without quotes by escaping these characters, so the double quotes are not an essential part of the Tcl language, but they make lots of things sooo much easier to code.

Otherwise, double dereferencing should be made so awkward.

As in the algorithm types I most frequently code in Tcl/Tk constructed names and double dereferencing are frequently the way to go, let me repeat myself: now ***" ... variable and command names you never in your right mind would think of using are accepted in definitive places just to prevent (rather make very awkward) the use of constructed names."*** That says it all.

(presumably) NJG: BTW, I don't think I suggested the dropping of the present array notation above!

Lars H: I never claimed you did. I remarked that your suggested juxtaposition as array indexing would not, given your suggested equivalence of $<string> and <string>, in the case of double indexing work the way that you indicated you wanted it to work.

FB: IMHO, as much as I love Tcl, if there is a point that is inconsistent, it's the variable substitution syntax. For example, the ${name} form does not follow the same brace matching rules as for rule [6]. I.e. the following code produces surprising results:

```
% set {var} foo
% puts ${{var}}
can't read "{var": no such variable
```

That's because the parser stops at the first close brace, whereas rule [6] balances them properly. Worse, there isn't any way to work around this limitation other than using set directly. I understand that this is the intended behavior as per rule [8], but I fail to see the overall consistency. And this is where the "$ is a shortcut for set" mantra fails.

This is one of the issues I try to address with Cloverfield. See Cloverfield - Tridekalogue rule [8] (and please try to make abstraction of the other propositions such as rule [6] and their implications to variable substitution), as well as allowing extended syntax for easier subscripting (see also Tcl 9.0 WishList #72 and Better Arrays for Tcl9). In short, I propose that $ be a prefix to words (or parts of words) that would be normally parsed and substituted, and whose resulting value would designate the variable name. This name part could also be suffixed by an "index" part (e.g. array subscript enclosed between parentheses, like the current syntax). For example, the following syntaxes would be accepted:

```
$name                   # Same as with Tcl
${name}                 # Ditto
${{name}}               # Same as [set "{name}"]. Uses rule [6].
$"string with spaces"  # Same as with {} but using rule [4]
$[proc]                 # Same as [set [proc]]
```

The index part allows vector and keyed access semantics using an interface concept borrowed from Feather:

```
$name(foo bar)          # Same as [dict get $name "foo bar"]
$name{1 2 3}            # Same as [lindex $name 1 2 3]
```

There can be several index parts:

```
$name(foo)(bar)         # Same as [dict get $name foo bar]
$name{1}{2}{3}          # Same as [lindex $name 1 2 3], but redundant with $name{1
2 3}
$name{1}(foo)           # Same as [dict get [lindex $name 1] foo]
$name(foo){1}           # Same as [lindex [dict get $name foo] 1]
```

Unfortunately we need distinct syntaxes for vector (numerical index) and keyed accesses (strings) because EIAS, and dicts are also valid lists. Besides, I don't want to fall into the same trap as PHP.

These changes of course potentially break compatibility. However the case about brace matching still stands IMHO and should not cause any compatibility issue (I don't want to meet the kind of psychopathic maniac that would use braces in variable names).

NJG 2005-02-09: **Lars**, I really appreciate your effort but you too are not reflecting on what I am saying here i.e. that the rules should be modified to provide a consistent result with regard to the two types of dereferencing. As it stands the only true way to get the value of a defined variable is with set. Which is even a more awkward feature of Tcl than the expr. Both lay Tcl open to ridicule, which as you know kills.

MS: would be interested in seeing a concrete proposal for the modified rules that provide that *consistent* result.

DKF: I think you're way wrong, NJG, but just because I think that doesn't mean that I'm right. Hence if you want to continue this discussion, I urge you to produce a full proposal.

RHS: s/*consistent*/consistent and doesn't break up to millions of lines of existing Tcl code/

RS: set is indeed the original true way, and accepts any string as variable name. The $ parser was added as a convenience, and follows typical variable-naming rules ([A-Za-z0-9_]), if the name is not braced. That's part of its convenience, it "does what I mean" :)

```
puts this:$this,that:$that,etc...
```

I see no problem there.

Peter da Silva: RS is correct. The behavior of $ is deliberate, and should not be changed. There's only one thing in the behavior of $ that was accidental. This is the meaning of $(...)... and in practice this is interpreted as set "(...)". This was something I wanted to make use of right at at the beginning when Karl and I were designing hashes. I wanted

$(expression) to be similar to expr {expression} *except* that variables would not need to be preceded by '$'. Eg... $(a+1) would be expr {$a + 1}... bringing variables and expressions closer together.

SYStems: Basically I want to make sure I understand {*} right! I tried before reading its explanation about it, until I reread the first chapter from *Practical Programming in Tcl and Tk*. Tcl does grouping first substitution second, this means the result of a substitution cannot affect the number of words a command receive, so is {*} a trick to overcome this? So far it seems, so. {*} substitute that the result may or may not be several words!

I always wonder why expand, was not implemented as a command, but this is now more obvious, {*} affect the behavior of the Tcl parser, thus this new feature cannot be implementated as a command, it's a rule Tcl must always follow.

I wonder why no a more sensible syntax, like [:a b c:], or [| a b c |] or or something or that nature, two consecutive brackets of different types, anyway ... this is not very important

Also,not very important

would

```
cmd a{*}{b c} d {*}{e f}
```

result in

```
cmd ab c d e f
```

or what?

Lars H: The reasons not to use e.g. [: ... :] are

1. It already means something different (command substitution with a command named **:**). {*}$x was prior to 8.5 a syntax error.
2. It's unTclish. None of the Tcl syntax rules requires looking at more than one character at a time for determining what to do, but this would lend a special significance to a pair of characters.

Regarding your question, that should result in

```
cmd a{*}{b c} d e f
```

since {*} is only special at the beginning of a word; the same is by the way true for left braces and quotes too, so if you try the above you'll find that the b and the c end up in different arguments.

Terminology

escargo 2005-04-25: There are some aspects of terminology in these rules that I find *bothersome.*

quote - In the history of punctuation, *quoting* usually has been a punctuation convention that uses **opening** and **closing** *quotation marks.* These marks very from language to language, and even in American English and British English. They still have in common the notion of *balance* (under most circumstances). Similarly, in many programming languages, there is the notion of an *escape character* and an *[L1]escape sequence* . This being the case, I find the use of the backslash to be an instance of an *escape character* and not of something being *quoted.* Rule 6 uses the phrase "quoted with a backslash." I don't believe that backslashes quote anything. They are escape characters and they escape things.

I also propose changing the wording in certain places to change; for example in Rule 1: "unless quoted" would become "unless quoted or escaped".

forward references - Backslash substitution is referenced before it is defined. This might be due to a more analytic style versus a synthetic style. I prefer an order where terms are defined before they are used. I would move definition of things like backlash substitution before they are used.

word versus token - Is "" a word? Is "{}" a word? Is "." a word? All of these are character strings that can be called *words* in the processing of Tcl inputs. I would be more comfortable with the term *token* instead of *word* in these rules. (Maybe it's my compiler-writing training.)

end of file - An end of file is a command terminator. Perhaps this was not thought to be worth mentioning, but I was experimenting to see if a file that ends before a newline would be recognized as complete or not; it did seem to be.

That's probably enough for now.

Lars H:

Re quoting: Not all languages use distinct opening and closing quotation marks. In Swedish, the rule has rather been to *always* use the same character to open as to close! (For guillemet quoting there is a modern tradition to quote as »citera«, i.e., the German style, but otherwise it's the \u201D both for opening and closing.)

Re tokens: I think there's a point in *not* using this term, because Tcl's syntax really isn't like that of languages where you tokenise code. Tcl *can* be understood on a character-by-character basis, which is only very rarely the case with other languages -- they instead have to be explained in terms of tokens.

Re end of file: This is really a special case of "end of script". Does it need to be pointed out that a command ends when the script it is part of ends?

escargo: The American English opening and closing quotation marks we are using (" and ") are not distinct. (Although different types of quoting could use symmetrical marks, it is clearly not required.) The point is that there two marks, one at each end. So if there is only one mark, then it is not *quoting*.

Tcl does tokenize code, but it calls the pieces words even if they don't have any letters. I know it's my own preference, but I expect something called a word to be made up of letters, and when it isn't it *grates* on my metaphorical nerves.

If a line does not end with a newline, can it be a valid statement or command? It's a *corner case* that I would like to see covered somewhere (even if not in the sacred succinct description here).

MS: (not 100% sure I understood the question, still trying an answer). Of course it can be valid. The rules do not mention *lines* at all - only that naked *newline characters* are command separators and comment terminators.

escargo: Things can get confusing where there are *separators* and *terminators*. For example, I think PL/I used semicolons as statement terminators; Pascal used semicolons as statement separators. If a newline was a statement terminator, then a file that ended without a newline at the end might be erroneous for the last statement.

Duoas 2008-10-28:

just because some of the above questions on terminology are left unanswered)

A **quote** has nothing to do with punctuation or balance. It has to do with how the object is treated. For example, if I were to quote Hamlet: "Alas! poor Yorick! I knew him, Horatio" (act 5, sc. 1, lines 202-3), one would expect to be able to verify that the text is *unchanged*. If I say "Alas poor Yorick! I knew him well!", then I have failed to quote it, since Hamlet never said that. (Another favorite: "Beam me up Scotty!")
So to 'quote' a word means that it is not modified. The mechanism for marking a quote can be anything.
I think the phrasology comes from functional languages (LISP or Scheme or some other variant?)

A **word** is a grammatical term to indicate a distinct unit of meaning. It is composed of more atomic units. In English, words are combinations of letters, which have no individual value (in modern language, at least). In hardware, a WORD is often composed of two BYTES, but it is considered one value, not two. In computer languages, a word (if it is so called, as it is in Tcl) represents a separable unit of meaning. Hence, in Tcl the string

```
puts {Hello world!}
```

consists of two words: puts and Hello world!. The value (or meaning) of the word depends on context. In contrast, a *token* is not necessarily a *word*.

As for terminators, end of media is always end of content. Does that really have to be spelled out in the Rules?

LV: the definition of "letter" in the section on valid characters (that can be used in a variable name without brace quoting) should be clarified, IMO. I've seen tutorials that say that the word means 7 bit ASCII alphanumeric characters. I've also seen, here on the Wiki (for instance, on What kinds of variable names can be used in Tcl), developer claims that they have tried using things like umlauts in variable names without needing the braces.

So, some clarification is in order, in my opinion.

---

wdb: The new expansion rule with {*} enforces us to rewrite the 11 rules. Let us take this as an occasion to find some completely new description.

Here my proposal for replacement of 11 rules.

- I have optimised by the rule KISS.
- I have tried to avoid forward references in the description, only backwards.
- I have written it as if I wanted to write a parser for Tcl (which is not the case).

The {*}expansion is described as step in substitution before introducing the terminus *list*. This way there is no needs to formulate some exception rule. I don't like exceptions at all in basic explanations.

Note that the description of \ at line end is **not** where the description of \ as escape sequence is but instead where in the program flow I would suppose it. My intention was: **Straight ahead, Sir!**

---

aspect: some behaviour I just picked up on that I find confusing, and would like to make a note of:

```
% puts "\}"
}
% puts "\{"
{
```

.. as expected. But then:

```
% puts {\{}
\{
% puts {\}}
\}
```

.. and for completeness:

```
% puts {\\}
\\
```

This is an interaction of rules [6] and [9], but in my eyes an inconsistency with:

```
% puts {\
}

%
```

In the case of backslash-newline, the backslash "escapes" or "quotes" the newline and is removed (rule [9]). In the case of backslash-brace, the backslash escapes the brace but is **not** removed (rule [6]). This inconsistency hurts as there's no way to embed a lone { or } in a {}-quoted string!

It gets even more mind-melting when you try and express what's going on using expr:

```
% expr {{\{}=="\\{"}
```

.. is not a complete command. You need instead to type:

```
% expr {{\{}=="\\\{"}
1
```

.. which hurts my brain.

AMG: I discuss the strangeness of backslash-newline earlier on this page.

Script

A Tcl script is a string.

Separator

If that string has a non-escaped semicolon (;) or newline, then this character serves as separator such that another script can follow.

Substitution

Before execution, the interpreter substitutes the string from left to right as follows:

1. If a line is terminated by the backslash character (\), then the following line is taken as continuation of current line where the backslash and following spaces are replaced by a single space.
2. If the line starts with zero or more spaces and the character (**#**), then the line is taken as comment. It is not executed.
3. If there is an opening brace (**{**) in the string, then finding the matching counterpart (**}**) has precedence above the separator, i. e. the group can contain newlines and semicolons. This section is protected against further substitution.
4. If a section is a double quote (**"**), then finding the closing double quote has precedence above the separator, i. e. the group can contain newlines and semicolons.

5. If a section of the string is enclosed in brackets (**[...]**), then the inner area is recursively executed as an embedded script, and the bracketed section is replaced by the result of the inner script.
6. If there is a backslash (\), then it escapes the following character as follows: a -- bell; b -- backspace; f -- form feed; n -- newline; r -- carriage return; t -- tab; v -- vertical tab; ooo -- octal number; xhh -- hex number; uhhhh -- unicode character; all other chars escape to themselves, especially braces and brackets described above.
7. If there is a character sequence **{*}** immediately followed by a non-space character, then the sequence **{*}** and the following group is replaced by the elements of that that group taken as list (without grouping characters).

After these steps, the string is transformed to a properly-formatted list, the first word of which is taken as procedure name, and the remaining arguments are taken as arguments (Polish notation). Then, the procedure belonging to that name is executed.

List

A string is a proper list if it contains words where:

1. A word consists either non-space characters or a group delimited by double quotes (**"**) or properly-balanced braces {...}.
2. A word is separated from its neighbours by space characters.

Note that if a word enclosed by double quotes ("...") or braces ({...}) is followed by a non-space character, then the string is not a properly-formatted list. If there is a neighbour, then the delimiting space character is mandatory.

CMcC can never remember whether he means endekalog or dodekalog, so uses *dekalog to mean both of them, and more (please refer to context for actual meaning.)

Proposal: More Consistent Variable Substitution

PYK 2014-12-24:

The following is a proposal to modify the variable substitution rules, resulting in three new behaviours:

1. Braced variable names are interpreted in accordance with the rules for braced words.
2. True multidimensional array indexing is supported.
3. Namespace paths are represented by standard lists rather than components delimited by two or more colons (::).

The further implication of this proposal is that commands in namespaces would also be represented by lists rather than by words delmited by two or more colonts (::). This proposal, of course, is not backwards-compatible, not by a long shot.

Variable Substitution

$*name*
When $ occurs in a word, and is followed by a sequence of standard ASCII characters in the ranges **0-9**, **A-Z**, and **a-z**, or underscore (**_**), that sequence is the name of a variable, the value of which replaces the $ and the variable name in the word.

$*name*(*path*)
When *name* is immediately followed by (, it names a variable. Subsequent characters up to the matching) constitute *path*, which is subject to command substitution, variable substitution, and backslash substitution, and the value of which must be a properly-formatted list. The last item in the list is the name of an item in the array or value, and the previous items of which specify a path into the array or value. The value of the item specified by *path* is substituted into the word in place of the $, variable name, parenthesis, and list.

${*names*}
If $ is immediately followed by an {, the characters up to the matching } are interpreted according to the rule for braced words, and must be a properly-formatted list, the last item of which specifies the name of a variable, and the previous items of which specify the namespace path of that variable. The matching } may be immediately followed by non-whitespace. No substitutions are performed on any characters between the parentheses that enclose an array path in the last item in the variable name.

Historical

The following information is obsolete, but retained for historical purposes

Documentation for Argument expansion

MS 2003-10-26: The acceptance of *Argument expansion with leading {*}* requires a new rule, and slight editions of the older ones.

This page holds a proposed wording for the new Tcl(n) manual page. I'd appreciate comments and suggestions.

escargo 2005-04-22: What about corrections for spelling and grammar? Should those be made in line?

escargo 2005-04-25: Hearing no objection, I'm going to start fixing spelling and grammar. I have some more editorial and conceptual issues, but I will start some discussion of those issues at the bottom.

LV: how many of the following changes have submitted for inclusion into Tcl 8.5?

Required Changes

- add new rule [5], shift old rules [5]-[11] up by one
- modify rule [6] **Braces** to acknowledge the {*} exception

- modify rule [12] **Substitution and word boundaries** to acknowledge the {*} exception

Clarifying Rule 2

See revision 148 of this page for the previous contents of this section, which describe a proposed modification of Rule 2 that has been merged into the official documentation.

---

kaelbling - 2015-12-22 10:02:02

Perhaps "word expansion" would be a better term than "argument expansion".

From page 4 of *Tcl and the Tk Toolkit*, 2nd ed.: "Each command consists of one or more *words*.... The first word of each command is the name.... The other words are *arguments*....

Since {*} can be used to supply the command name as well as the arguments it is not strictly for arguments.

```
{*}{puts happy}
```

Furthermore "words" are an entry in the introductory glossary, but "arguments" are not.

(This comment was inspired by the "script substitution" vs. "command substitution" paragraph.)

pyk 2019-11-24: "command expansion" is another option, since it is the command that is growing.

---

kaelbling - 2015-12-22 10:20:18

Unhappiness with variable substitution syntax

... or with the set command -- one of them is inconsistent.

```
% set arr(f)) true
% puts $arr(f))
can't read "arr(f)": no such element in array

% set arr(g()) true
% puts $arr(g())
can't read "arr(g()": no such element in array

% set arr(a;b) true
can't read "arr(a": no such variable
% puts $arr(a;b)
can't read "arr(a;b)": no such element in array

% array names arr
g() f)
```

RLE (2015-12-22): Not inconsistent, just a misunderstanding. Variable names can consist of nearly any string in Tcl. However, the confusion you have above is due to interactions between the names you have chosen and one or both of the Tcl parser and/or the parser for "$" expansion. If you escape the characters that both use as 'special', all of your names above work perfectly:

```
% set arr(f)) true
true
% puts ${arr(f))}
true
% set arr(g()) true
true
% put ${arr(g())}
true
% puts ${arr(g())}
true
% set {arr(a;b)} true
true
% puts ${arr(a;b)}
true
% array names arr
g() {a;b} f)
```

The name "arr(f))" interacted badly with the "$" parser - because the "$" parser uses parenthesis to denote 'array' and the parser saw a matched pair after "arr(f)" and stopped parsing. But as you had not set that array entry, there was no array entry with a name of "f". Your second one failed for similar reasons (although it looks from #2 like the "$" parser does not look for nested parenthesis). But the simple explanation is "(" and ")" are 'special' to "$", be careful with them when using them in ways "$" does not expect.

Your third "set arr(a;b) true" failed because the tcl parser sees ';' as special. It means "command ends here". So what the parser found was these two commands smashed onto one line:

```
set arr(a
b)
```

The first attempts to retrieve the value in the variable "arr(a", there is no value there, so you get a no such variable error. At which point the Tcl shell stops and waits for more input from you.

Alternately, if you do want to use 'special' characters in variable names, you can also use set to both create and retrieve their values:

```
% set arr(f)) true
true
% set arr(f))
true
% set arr(g()) true
true
% set arr(g())
true
% set arr(a\;b) true
true
% set arr(a\;b)
true
% parray arr
arr(a;b) = true
arr(f))  = true
arr(g()) = true
```

PYK 2015-12-22: Although kaelbling's examples can all be fixed using variable brace notation, there is an asymmetry between variable substitution syntax and set. The variable substitution rule doesn't guarantee that all variables can be accessed using its syntax, and indeed, they can't. In the following example, there's no way to get at the value of var} using variable substitution:

```
set var\} hello
```

The proposal earlier on this page, **More Consistent Variable Substitution**, would remedy that.

---

jcd - 2017-07-20 18:41:06

The description of comment above is incorrect.

The claim:

```
    Comment
      If # is the first character in the name of a command,
      and is not otherwise quoted, the characters up to the
      next newline are a comment. No command execution is
      attempted, and no characters in the line are
      interpreted, except that a newline escaped with \
      is ignored.
```

To see that this is not true, consider:

```
   proc foo {} {
     # bar }
     puts baz
   }
```

Evaluating this prints *baz* and then reports *invalid command name "}"*.

Hence, the *}* character is **not** in fact ignored, even though it comes before the newline.

PYK 2017-07-20: Your interpretation is incorrect. The error message is complaining about the brace after puts baz. That # character is simply a literal character enclosed in a braced word. One of the subtle ways to get into the wrong frame of mind about Tcl syntax is to start thinking of things in terms of code blocks, which do not exist in Tcl syntax. There are only strings.

RKzn 2017-07-20: So it wouldn't hurt if someone clarified why both

```
proc foo1 {} {
    # bar
    puts baz
}
```

and

```
proc foo2 {} {
    # bar \}
    puts baz
}
```

do not raise the same error. I _think_ it is because the initial example is actually, the same as:

```
proc foo {} {
    # bar
}
puts baz
}
```

that is, a "empty" proc plus a puts command outside that proc plus a } which cause the error (as PYK said). All that because the rule about braces "captures" that } to pair up with the initial {, so the comment is "bar", not "bar }". And then, to further confuse things, we get fooled by the (erroneous) indentation.

EMJ 2017-07-20: Think like the Tcl interpreter.

proc is expected to be a command, and what follows will be its arguments. First, *foo* which will be the name of the procedure being defined. Then *{}* will be the list of formal arguments to the procedure being defined. The next *{* starts the third argument to *proc* (the procedure body) which will, by rule 6, be ended by the next matching unescaped *}*. The characters between the *{* and *}* are just a string, and are not evaluated in any way at this point (that's what braces do - rule 6).

The matching *}* is therefore the one after *bar*. In the next line, *puts* is expected to be a command. Since it is, it is evaluated and the string *baz* is output. The next line after that is '}' which is expected to be a command, but there is no such command, hence the error.

The procedure *foo* **has** been defined, and if you call it, it will do nothing because

```
% info body foo

 # bar
%
```

i.e. at this time, the # is expected to be a command but by rule 10 starts a comment so that line **of the procedure body** will be ignored.